

# Framebuffers/ Renderbuffers and Porting WebGL/ OpenGL

## Advanced Graphics Programming GD2P04

## Framebuffers

- Up to this point, all of our discussion regarding buffers has focused on the buffers provided by the windowing system as requested by user.
- `glutInitDisplayMode(`
- `GLUT_DEPTH | // depth buffer`
- `GLUT_DOUBLE | // double buffer`
- `GLUT_RGBA); // color buffer`

# Framebuffers

- Color buffer for writing color values,
- Depth buffer to write depth information and
- Stencil buffer that allows us to discard certain fragments based on some condition.
- The combination of these buffers is called a *framebuffer*

# Framebuffers

- Using framebuffer objects, you can create our own framebuffers.
- Use their attached renderbuffers to minimize data copies and optimize performance
- Framebuffer objects are quite useful for performing *off-screen-rendering*, updating texture maps

# Framebuffers

- Framebuffer that is provided by the windowing system is the only framebuffer that is available to the display system of your graphics server---that is, it is the only one you can see on your screen.
- By comparison, the framebuffers that your application creates cannot be displayed on your monitor; they support only off-screen rendering.

# Allocating and Binding FBO

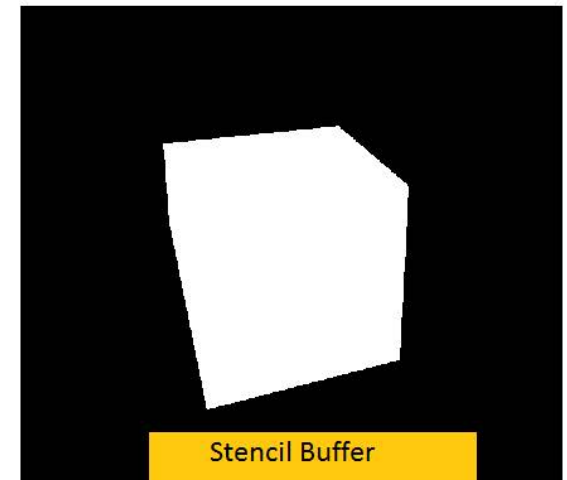
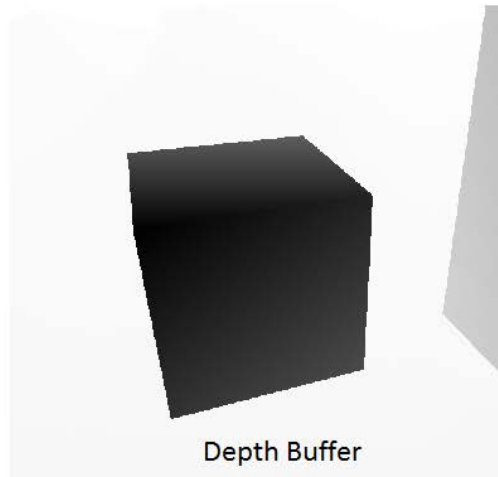
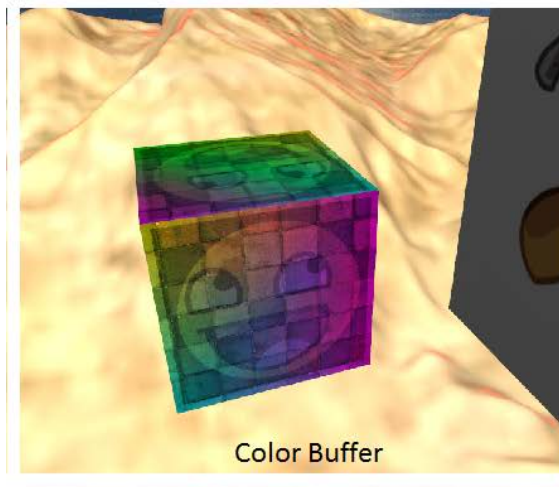
- Generating framebuffers
  - glGenFramebuffers(GLsizei n, GLuint \*ids);
- Binding FBO
  - glBindFramebuffer(GLenum target, GLuint framebuffer);
- Target
  - Specifies a framebuffer for either reading or writing.
  - GL\_DRAW\_FRAMEBUFFER, specifies the destination framebuffer for rendering.
  - GL\_READ\_FRAMEBUFFER, specifies the source of read operations.
  - GL\_FRAMEBUFFER for target sets **both** the read and write framebuffer bindings to framebuffer

## Texture as a target attachment

- Whatever is rendered to the framebuffer can be used as a texture.
- To see general output or used later in the pipeline.

```
glFramebufferTexture2D(GLenum target, //type of target  
    GLenum attachment, //color, or depth or stencil  
    GLenum textarget, // type of texture 1D, 2D or 3D  
    GLuint texture, // texture ID  
    GLint level); //mipmap level
```

# Color, Depth or Stencil Attachment





## Creating, binding and Attaching

```
GLuint framebuffer;  
glGenFramebuffers(1, &frameBuffer);  
glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);  
  
//Attach texture to framebuffer object  
glFramebufferTexture2D(GL_FRAMEBUFFER, //target buffer  
GL_COLOR_ATTACHMENT0, //attachment, could be  
                        //GL_DEPTH_ATTACHMENT or  
                        //GL_STENCIL_ATTACHMENT  
GL_TEXTURE_2D, //texture target type  
renderTexture, //texture  
0); // level
```

## Depth Attachment

- To attach a depth attachment we specify the attachment type as `GL_DEPTH_ATTACHMENT`.
- The texture's format and internal format type should be `GL_DEPTH_COMPONENT` to reflect the depth buffer's storage format.

# Creating Texture

```
GLuint renderTexture;  
glGenTextures(1, &renderTexture);  
glBindTexture(GL_TEXTURE_2D, renderTexture);
```

```
glTexImage2D(GL_TEXTURE_2D,  
0, GL_RGB,  
1280,720  
0, //border  
GL_RGB, //format  
GL_UNSIGNED_BYTE, //data type , NULL);
```

- For this texture, we're only allocating memory and not actually filling it

# Render Buffers

- Renderbuffers are effectively memory managed by OpenGL that contains formatted image data.
- The data that a renderbuffer holds takes meaning once it is attached to a framebuffer object.
- It stores its data in OpenGL's native rendering format making it optimized for off-screen rendering to a framebuffer.
- Renderbuffer objects are generally write-only, thus you cannot read from them.
- Since depth tests cannot be performed on a texture we need the render buffer to do depth and stencil values for testing.

## Render Buffers

- Before you can attach a renderbuffer to a framebuffer and render into it, you need to allocate storage and specify its image format.
- This is done by calling *glRenderbufferStorage()*
- Then attach the renderbuffer to the framebuffer which is done by calling *glFramebufferRenderbuffer()*

## Creating and Binding RBO

```
GLuint rbo;  
glGenRenderbuffers(1, & rbo);  
glBindRenderbuffer(GL_RENDERBUFFER, rbo);  
glRenderbufferStorage(GL_RENDERBUFFER, // must be  
    GL_DEPTH24_STENCIL8, //use as depth - stencil buffer  
    1280, 720) //viewport width and height;  
glFramebufferRenderbuffer(GL_FRAMEBUFFER, //target  
    GL_DEPTH_STENCIL_ATTACHMENT, //attachment  
    GL_RENDERBUFFER, //renderbufferTarget  
    rbo); // render buffer
```

## Framebuffer Testing

- Check if framebuffer is created and texture and renderbuffer are attached

```
if (glCheckFramebufferStatus(GL_FRAMEBUFFER)
    != GL_FRAMEBUFFER_COMPLETE) {
    cout << "ERROR::FRAMEBUFFER:: Framebuffer is
    not complete!" << endl;
}
```

## Using the framebuffer

- Bindframebuffer
- Clear color, depth and stencil bit
- Draw the object
  - Draw skybox
  - Draw cube, etc.
- Bind the Texture and use the texture as shader resource and draw it on a quad.

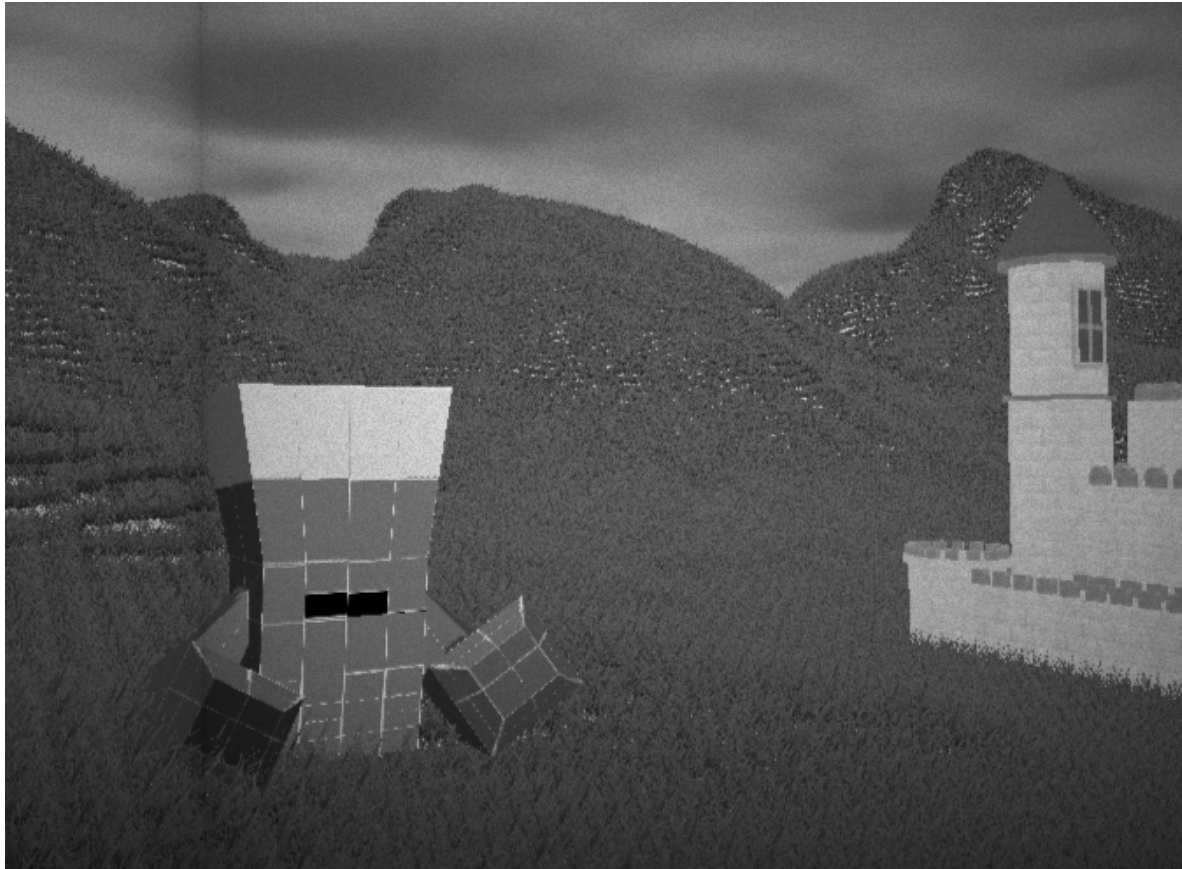


# Code

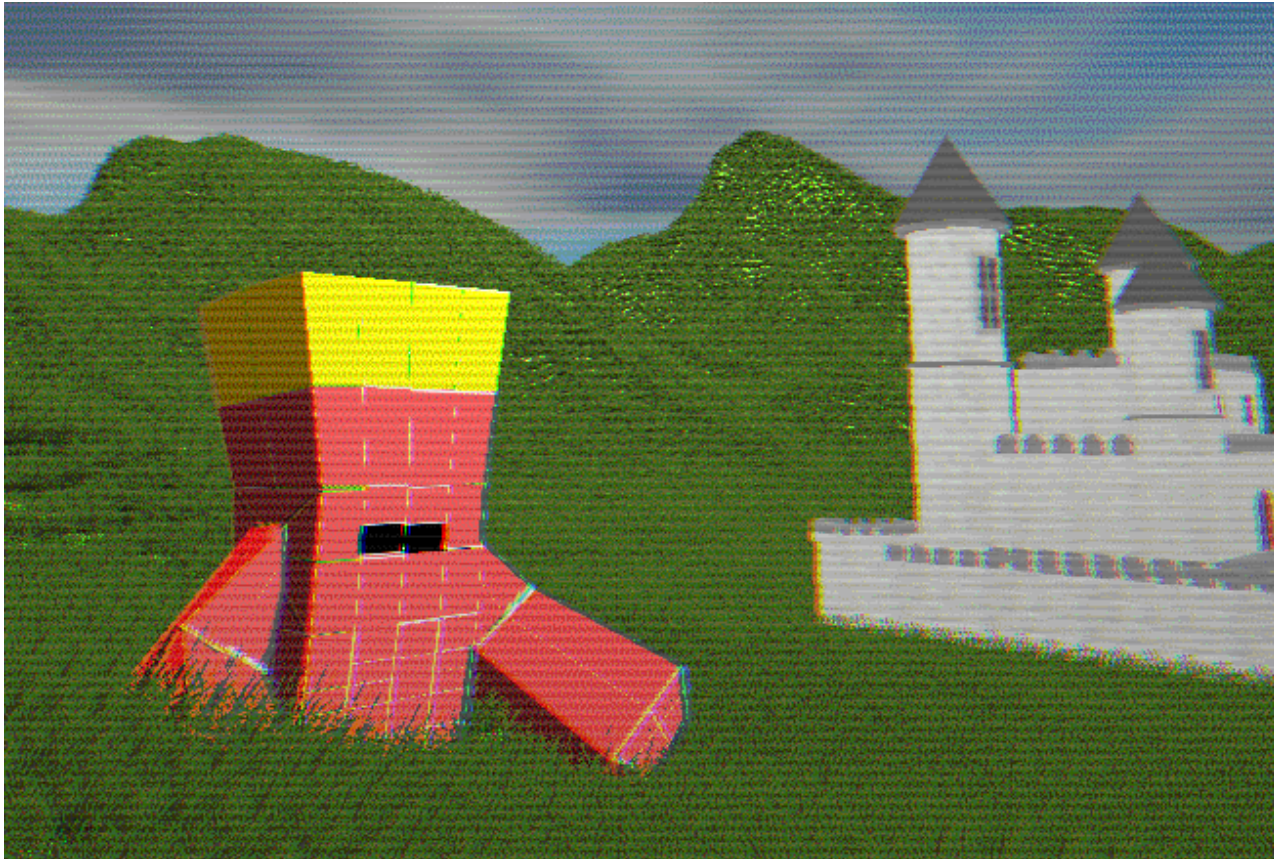
```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);  
glClearColor(1.0f, 1.0f, 0.0f, 1.0f);  
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);  
  
Cube->draw();  
  
glUseProgram(program);  
glActiveTexture(GL_TEXTURE0);  
glUniform1i(glGetUniformLocation(program, "renderTexture"), 0);  
glBindTexture(GL_TEXTURE_2D, renderTexture);  
  
glBindVertexArray(quadVAO);  
glDrawArrays(GL_TRIANGLES, 0, 6);  
glBindVertexArray(0);  
  
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

# Output

- Old movie scratch and pops

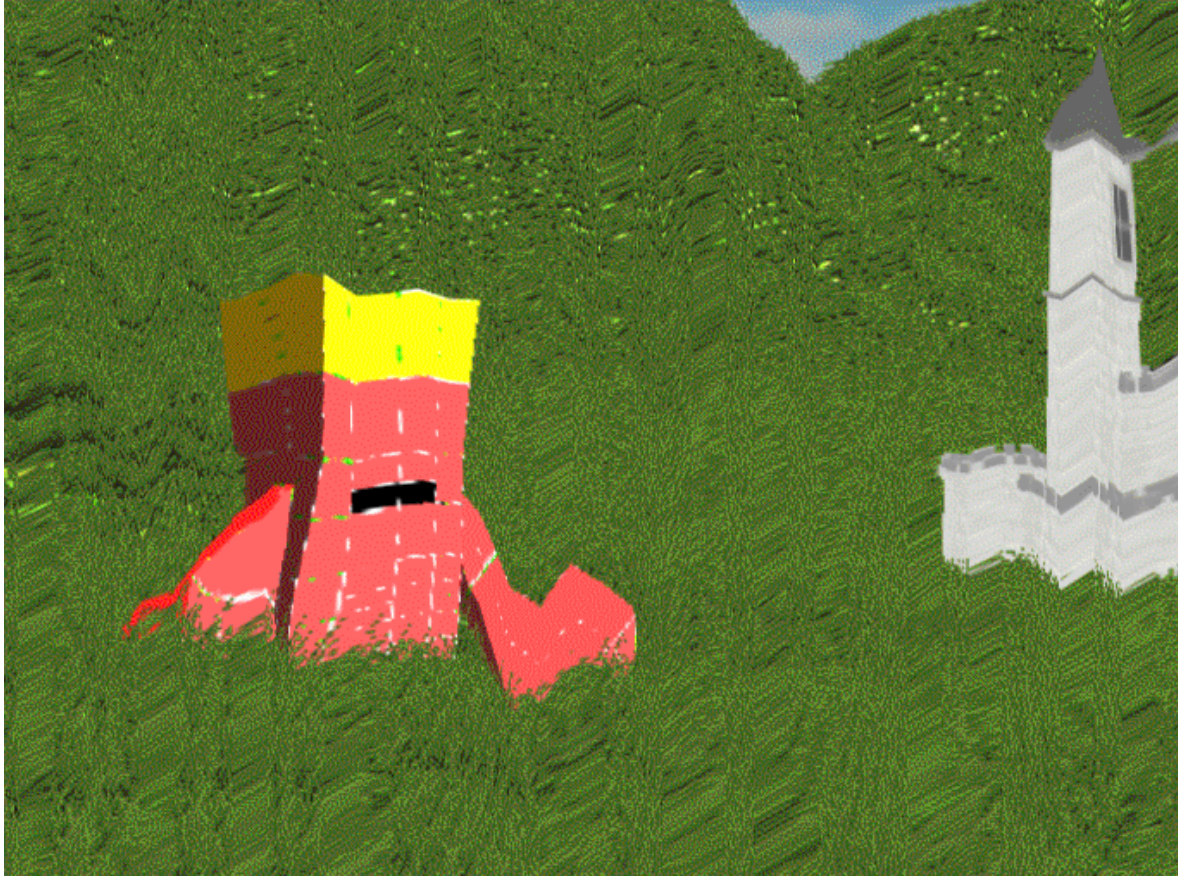


- Chromatical, scanlines, distortion



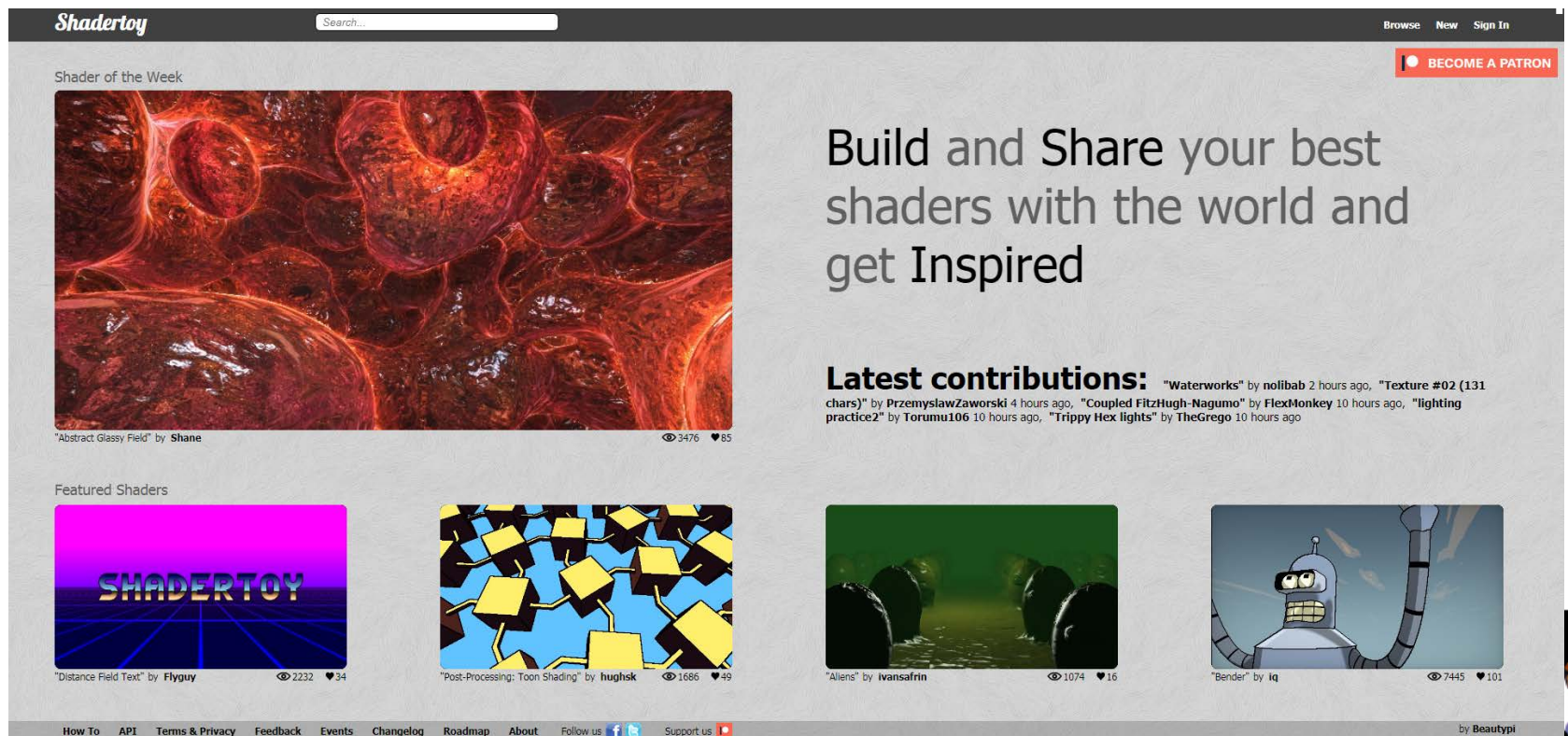


- Raining on your screen



# Porting WebGL/ openGL

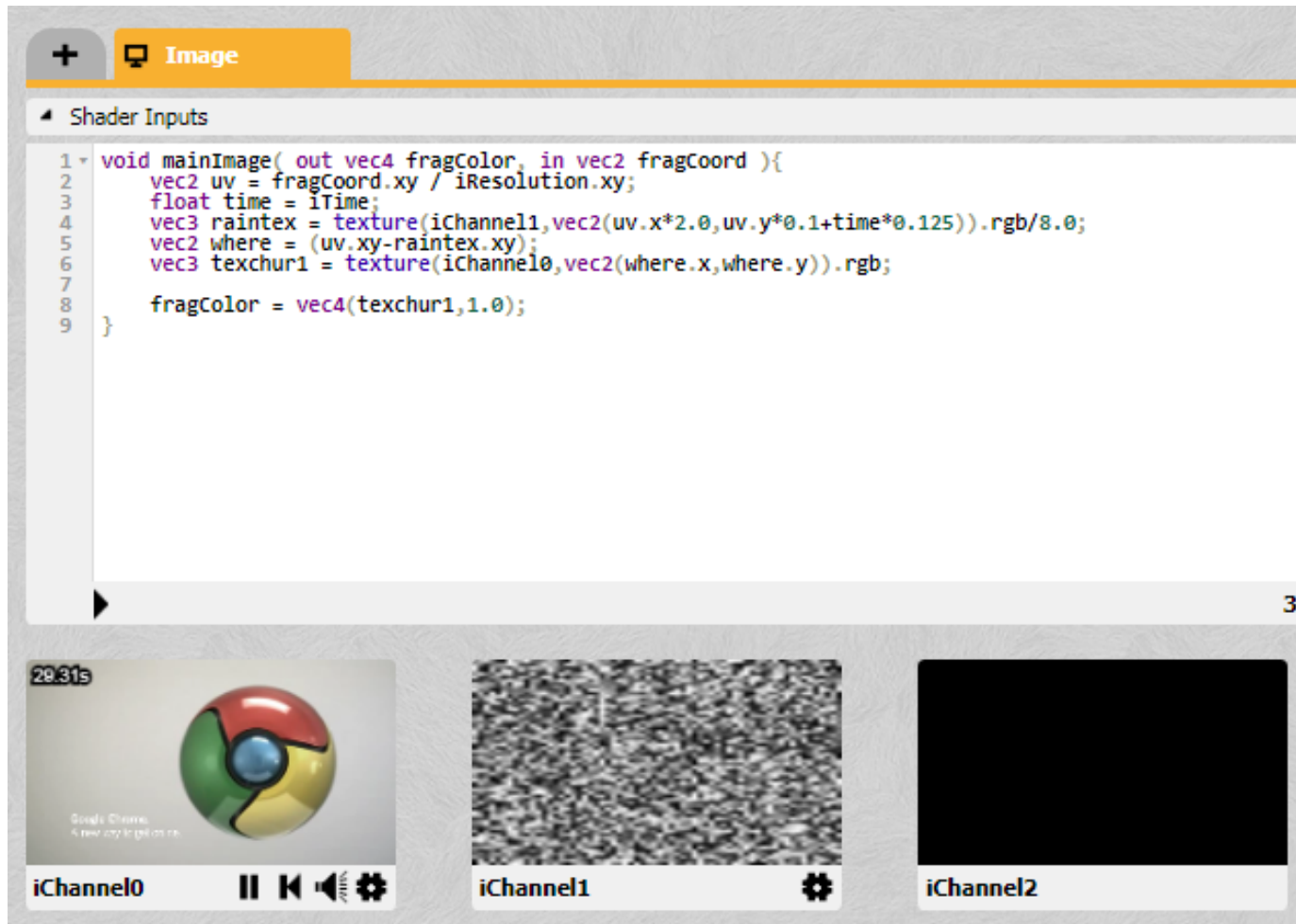
- <https://www.shadertoy.com/>



## Data type conversions

- fragCoord: vec2 texturecoordinate information
- iResolution: vec2 with width and height of screen
- varying are uniforms
- fragColor: vec4 output color
- iChannel0, iChannel1- textures passed in
- iTime - current time passed in as uniform float
- Slight adjustment to values might be required to get similar effect.

# Shadertoy code



The image shows the Shadertoy web application interface. At the top, there's a tab labeled "Image" with a plus icon. Below it, a section titled "Shader Inputs" contains a code editor with the following C++-like shader code:

```
1 void mainImage( out vec4 fragColor, in vec2 fragCoord ){
2     vec2 uv = fragCoord.xy / iResolution.xy;
3     float time = iTime;
4     vec3 raintex = texture(iChannel1,vec2(uv.x*2.0,uv.y*0.1+time*0.125)).rgb/8.0;
5     vec2 where = (uv.xy-raintex.xy);
6     vec3 texchur1 = texture(iChannel0,vec2(where.x,where.y)).rgb;
7
8     fragColor = vec4(texchur1,1.0);
9 }
```

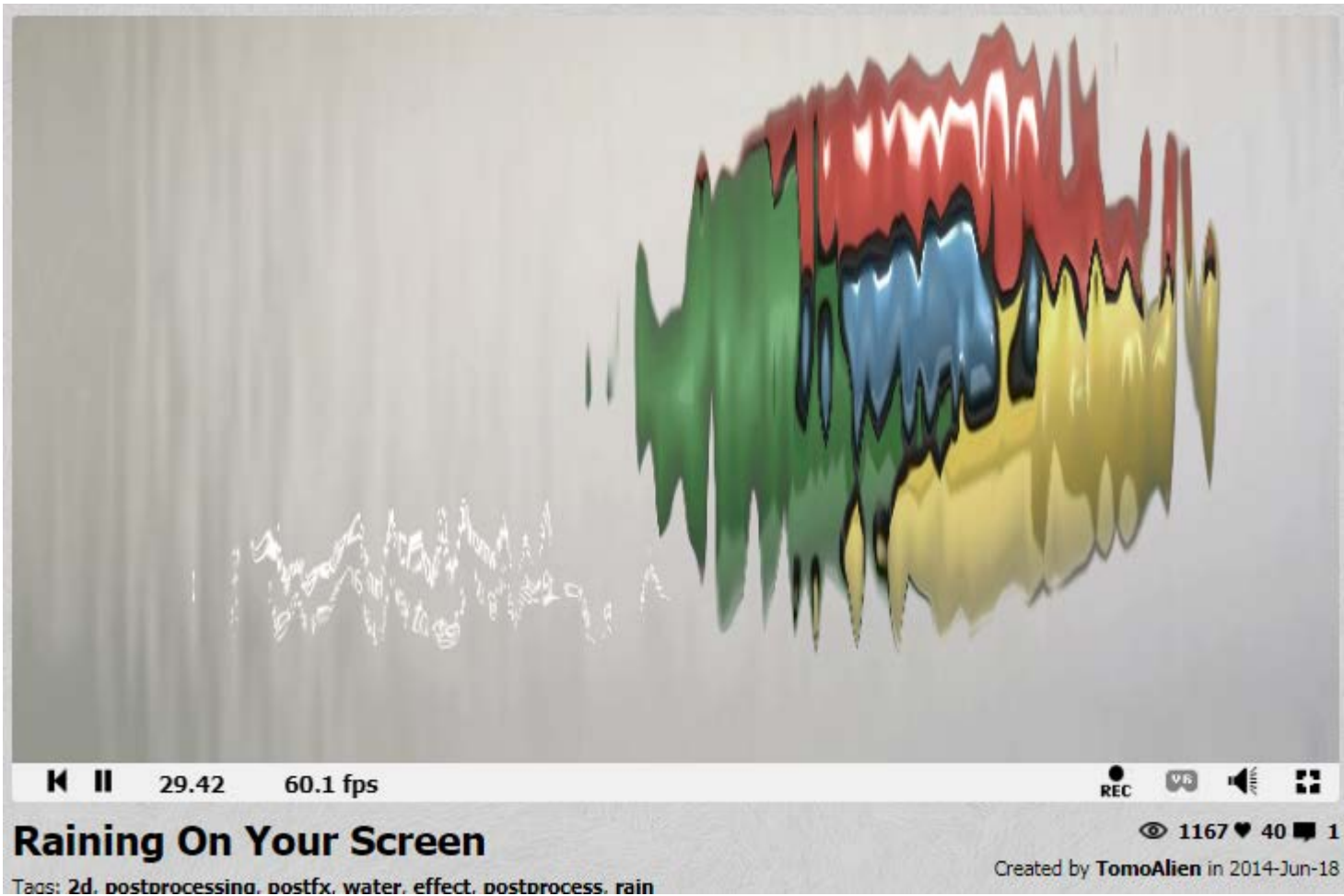
Below the code editor, there are three preview images:

- iChannel0**: A preview of the Google Chrome logo, with a timestamp of 29.31s and playback controls (play, stop, volume, settings).
- iChannel1**: A preview of a noisy texture, with a settings icon.
- iChannel2**: A preview of a solid black image, with a settings icon.

A small number "3" is visible in the bottom right corner of the main interface area.



# Shadertoy Output





## Excercise

- Create a framebuffer.
- Create a texture attachment.
- Render objects to it.
- Create a quad and apply render texture to it.
- Take a shader effect from Shadertoy and apply to the rendertexture.